



Universidad Nacional
Autónoma de México

Facultad de ingeniería



Profesor: M. I. Alberto Navarrete Hernández

Guadarrama Herrera Ken Bryan
Gómez Luna Brisa Mariana
Martínez Araujo Jesús Alonso
Montiel Avilés Axel Fernando

Grupo: 3

Trabajo Final

Para la realización del proyecto decidimos trabajar con una arquitectura RISC de 24 bits, donde contaremos con 10 registros, de los cuales, dos de ellos se deberán utilizar en las operaciones de lectura y escritura de la memoria principal y los ocho restantes se manejarán en las demás operaciones de la arquitectura:

Registros		
areg	0000	0
breg	0001	1
creg	0010	2
dreg	0011	3
ereg	0100	4
freg	0101	5
greg	0110	6
hreg	0111	7
ireg	1000	8
jreg	1001	9

Donde los registros ireg y jreg serán para lectura y escritura respectivamente.

En cuanto a los formatos de instrucción se pueden dividir en la siguiente clasificación dependiendo del tipo de operación a realizar:

a) Operaciones Aritméticas y lógicas:

Para este tipo de operaciones, contamos con la instrucción Suma, Resta, División, Multiplicación y Comparación, así como AND, OR, NOT, XOR, SHL, SHR, ROL, ROR Y TEST.

El formato de instrucción tiene la forma:

Operación-Resultado-Operando_1-Operando_2.

Donde el formato de instrucción final es el siguiente:

Opcode (5 bits)	Resultado (4 bits)	Operando1 (4 bits)	Operando2 (4 bits)	Relleno (7 bits)
--------------------	-----------------------	-----------------------	-----------------------	---------------------

Al ser una arquitectura tipo RISC, debemos de cumplir con una longitud fija en los formatos de instrucción, es por esto que acudimos a los bits de “relleno”, los cuales serán conformados por ceros, hasta cumplir con los 24 bits.

Ejemplos:

add greg, areg, breg

and dreg, ereg, freg

b) Operaciones de Lectura y Escritura:

Para este tipo de operaciones en la memoria principal, tenemos a los registros ireg y jreg para la lectura y escritura respectivamente, el formato de instrucción tiene la forma:

Operación-Register_destiny-[Register_address]

Operación-[Register_address]-Register_source

Donde los formatos de instrucción finales son los siguientes:

Opcode (5 bits)	Reg_Destino (4 bits)	[Reg_Address] (4 bits)	Relleno (11 bits)
--------------------	-------------------------	---------------------------	----------------------

Opcode (5 bits)	[Reg_Address] (4 bits)	Reg_Source (4 bits)	Relleno (11 bits)
--------------------	---------------------------	------------------------	----------------------

Ejemplos:

read areg, [dreg]

write [hreg], breg

read creg, [ireg]

write [jreg], freg

c) Operaciones entre registros generales

Para este tipo de operaciones contamos con la instrucción MOVE.

Donde el formato de instrucción final es el siguiente:

Opcode (5 bits)	Reg_Destino (4 bits)	Reg_Source (4 bits)	Relleno (11 bits)
--------------------	-------------------------	------------------------	----------------------

Ejemplo:

move breg, areg

En el caso de trabajar valores inmediatos, se tendrán 8 bits para representar el numero decimal, quedando el formato de instrucción:

Opcode (5 bits)	Reg_Destino (4 bits)	Valor inmediato (8 bits)	Relleno (7 bits)
--------------------	-------------------------	-----------------------------	---------------------

d) Saltos Incondicionales y Condicionales

Para los saltos incondicionales, así como los condicionales, hay que tomar en cuenta que se manejarán números sin signo.

Saltos incondicionales:

Para los saltos incondicionales se definió el siguiente formato de instrucción:

Opcode (5 bits)	Adress_etiqueta (8 bits)	Relleno (11 bits)
--------------------	-----------------------------	----------------------

Ejemplo:

```
move ireg, #direccion_destino
```

```
jmp ireg
```

Para los saltos condicionales hay que considerar las instrucciones:

JE: Salta a una etiqueta específica si la última operación de comparación estableció que dos operandos son iguales.

JNE: Salta a una etiqueta específica si la última operación de comparación estableció que dos operandos no son iguales

JA: Salta a una etiqueta específica si el operando anterior es mayor que el operando de destino (sin signo)

JAE: Salta a una etiqueta específica si el operando anterior es mayor o igual que el operando de destino (sin signo)

JB: Salta a una etiqueta específica si el operando anterior es menor que el operando de destino (sin signo).

JBE: Salta a una etiqueta específica si el operando anterior es menor o igual que el operando de destino (sin signo).

Ejemplo:

```
cmp areg, breg
```

```
jnd ireg
```

El formato de instrucción para saltos condicionales es el siguiente:

Opcode (5 bits)	Reg_Destino (4 bits)	Adress_etiqueta (8 bits)	Relleno (7 bits)
--------------------	-------------------------	-----------------------------	---------------------

OPCODE		
add	00000	0
sub	00001	1
mul	00010	2
div	00011	3
cmp	00100	4
and	00101	5
or	00110	6
not	00111	7
xor	01000	8
shl	01001	9
shr	01010	10
rol	01011	11
ror	01100	12
test	01101	13
read	01110	14
write	01111	15
move	10000	16
jmp	10001	17
je	10010	18
jne	10011	19
ja	10100	20
jae	10101	21
jb	10110	22
jbe	10111	23
jcd	11000	24

Códigos propuestos:

Programa1:

```

move areg, #15
move breg, #7
write [hreg], areg
read dreg, [hreg]
write [ireg], breg
read greg, [ireg]

```

Programa2:

```
move areg, #10
move breg, #5
move creg, #0
add dreg, areg, breg
sub ereg, dreg, creg
and freg, dreg, ereg
or greg, freg, creg
cmp hreg, greg, creg
je 10010, hreg
jne 10011, hreg
ja 10100, hreg
jae 10101, hreg
jb 10110, hreg
jbe 10111, hreg
```

Programa3:

```
move areg, #10
move breg, #5
move creg, #0
add dreg, areg, breg
sub ereg, dreg, creg
xor freg, dreg, ereg
shl greg, freg, #2
cmp hreg, greg, #20
je 10010, hreg
jne 10011, hreg
ja 10100, hreg
jb 10110, hreg
jmp 11000
```

Explicación del programa:

Definición de variables globales




```
4  # variable para guardar la ubicación del archivo
5  INPUT_FILE = "./asm.asm"
```



```
7  # lista con los nombres de los registros
8  # en orden
9  REG_CODES = [
10     "areg",
11     "breg",
12     "creg",
13     "dreg",
14     "ereg",
15     "freg",
16     "greg",
17     "hreg",
18     "ireg",
19     "jreg",
20 ]
```



```
22 # diccionario para guardar la linea donde
23 # se encuentra la etiqueta
24 LABELS = {}
```



```
26 # diccionario para guardar la ultima linea
27 # donde se uso el registro
28 DIR_REG = {
29     "areg": -1,
30     "breg": -1,
31     "creg": -1,
32     "dreg": -1,
33     "ereg": -1,
34     "freg": -1,
35     "greg": -1,
36     "hreg": -1,
37     "ireg": -1,
38     "jreg": -1,
39 }
```



```
41 # lista con las operaciones en orden
42 OP_CODES = [
43     "add",
44     "sub",
45     "mul",
46     "div",
47     "cmp",
48     "and",
49     "or",
50     "not",
51     "xor",
52     "shl",
53     "shr",
54     "rol",
55     "ror",
56     "test",
57     "read",
58     "write",
59     "move",
60 ]
```




```
62  # diccionario con las operaciones
63  # de salto de linea con la conversion
64  # a binario
65  JM_CODES = {
66      "jmp": "10001",
67      "je": "10010",
68      "jne": "10011",
69      "ja": "10100",
70      "jae": "10101",
71      "jb": "10110",
72      "jbe": "10111",
73      "jcd": "11000",
74  }
```

Función para leer archivo



```
77  def read_file(name) -> list[str]:
78
79      # recuperar todo el contenido del archivo
80      cont = []
81      with open(name, "r") as file:
82          file.seek(0)
83          cont = file.readlines()
84
85      # guardar todas las lineas en una lista
86      # quitando los saltos de linea
87      lines = []
88      for line in cont:
89          if not line == "\n":
90              lines.append(line)
91
92      return lines
```

Función para convertir a binario con “n” bits un entero

```
95 def binary_converter(n:int, bits:int):
96     # string para guardar el binario
97     binary_representation = ""
98
99     # copiar el numero entero a un aux
100    aux = n
101
102    # dividimos el numero entre 2
103    # y concatenamos el residuo de / 2
104    # mientras sea mayor a 0
105    while aux > 0:
106        binary_representation += str(aux % 2)
107        aux //= 2
108
109    # rellenar con 0s el string en caso
110    # de que en la conversion se necesiten
111    # mas bits
112    if len(binary_representation) < bits:
113        for _ in range(0, bits - len(binary_representation)):
114            binary_representation += "0"
115
116    # retornamos la cadena en binario
117    # volteada
118    return binary_representation[::-1]
```

Función para buscar etiquetas

```
191 def search_labels(file) -> None:
192     lines = read_file(file)
193
194     # indice de la linea
195     i = 1
196
197     # recorremos cada linea buscando una
198     # etiqueta
199     for line in lines:
200
201         entity = line.split(" ")[0]
202         entity = entity.replace("\n", "")
203
204         # si ej -> etiq:
205         if entity[-1] == ":":
206             # print(f"{i}: {entity}")
207
208             # guardamos la linea en donde
209             # esta la etiqueta
210             LABELS[entity[:-1]] = i
211
212             # aumentamos el numero de linea
213             i+=1
```

Función para convertir las instrucciones con operaciones (sin saltos de línea)

```
121 def op_instruction(instruction:str, line:int) -> str:
122
123     # limpiar instrucción
124     instruction = instruction.replace("\n", "")
125
126     # separando por espacios la instruccion
127     entities = instruction.split(" ")
128
129     # buscar el indice de la operacion
130     op = OP_CODES.index(entities[0])
131
132     # convertir el indice de la operacion a
133     # binario de 5 bits
134     bin = binary_converter(op, 5)
135
136     # iteramos el resto de registros u operaciones
137     # quitando la primer operacion
138     for entity in entities[1:]:
139
140         # quitando las comas
141         entity = entity.replace(",", "")
142
143         # si lo que esta despues de la operacion...
144
145         # es un registro
146         if entity in REG_CODES:
147             # guardamos la ultima linea donde se uso
148             # el archivo
149             DIR_REG[entity] = line
150
151             # buscamos el indice del registro
152             # para despues convertirlo a binario
153             reg = REG_CODES.index(entity)
154             bin += " "
155             bin += binary_converter(reg, 4)
156
157         # es una operacion de lectura/escritura
158         # con un registro
159         elif "[" in entity:
160
161             # limpiando el registro
162             entity = entity.replace("[", "")
163             entity = entity.replace("]", "")
164
165             # guardamos la ultima linea donde
166             # el registro se uso
167             DIR_REG[entity] = line
168
169             # recuperamos la ultima linea donde se uso
170             # el registro para convertirla a binario
171             dir = binary_converter(DIR_REG[entity], 4)
172             bin += " "
173             bin += dir
174
175         # es una asignacion con valor inmediato
176         elif entity[0] == "#":
177             bin += " "
178             # convertimos a binario quitandole
179             # el simbolo "gato"
180             bin += binary_converter(int(entity[1:]), 8)
181
182     # retornamos la instruccion convertida a binario
183     return bin
```

Función para convertir las instrucciones con salto de línea

```
186 def jm_instruction(instruction:str) -> str:
187     # limpiamos la instruccion
188     instruction = instruction.replace("\n", "")
189
190     # separamos por espacios
191     entities = instruction.split(" ")
192
193     # recuperamos el binario de la operacion
194     # de salto del diccionario
195     bin = JM_CODES[entities[0]]
196
197     # si la instruccion tiene mas de 2
198     # "entidades", ej: jbe 10111, hreg
199     if len(entities) > 2:
200
201         bin += " "
202         # quitando comas
203         label = entities[1].replace(",", " ,")
204
205         # recuperamos la linea donde se encuentra
206         # declarada la etiqueta y la convertimos
207         # a binario
208         bin += binary_converter(LABELS[label], 8)
209
210         bin += " "
211
212         # recuperamos el indice del registro que
213         # se usa para convertirlo a binario
214         reg = REG_CODES.index(entities[2])
215         bin += binary_converter(reg, 8)
216
217     # si la instruccion tiene solo 2
218     # "entidades", ej: jbe 10111
219     else:
220         bin += " "
221         # solo recuperamos la linea de la etiqueta
222         # para convertirla a binario
223         label = entities[1]
224         bin += binary_converter(LABELS[label], 8)
225
226     return bin
```

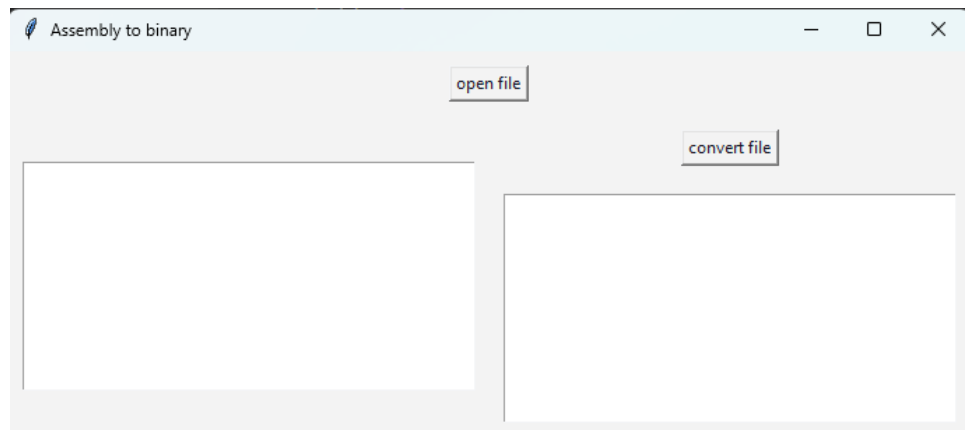
Función principal de conversión

```
254 def conversor(file) -> list[str]:
255
256     # declaramos una lista que es donde se van
257     # a guardar todas las instrucciones en binario
258     ans = []
259
260     # buscamos las etiquetas
261     search_labels(file)
262
263     # leemos de nuevo el archivo
264     lines = read_file(file)
265
266     # string para guardar el binario
267     out = ""
268
269     # indice de la linea actual
270     i = 1
271
272     # leemos cada linea del archivo
273     for line in lines:
274
275         # separamos la instruccion por espacios
276         # para recuperar la primer "entidad"
277         entity = line.split(" ")[0]
278
279         # quitando saltos de linea
280         entity = entity.replace("\n", "")
281
282         # print(entity)
283
284         # string para guardar la
285         # instruccion
286         ins = ""
287
288         # si la operacion no es una de
289         # salto de linea
290         if entity in OP_CODES:
291
292             # convertimos la operacion a binario
293             # y la concatenamos
294             ins = op_instruction(line, i)
295             # print(f"{i}: {ins}")
296             out += ins
297
298         # si la operacion es de salto de linea
299         elif entity in JM_CODES:
300             # convertimos a binario y concatenamos
301             ins = jm_instruction(line)
302             # print(f"{i}: {ins}")
303             out += ins
304
305         # vamos aumentando el indice de la linea actual
306         i += 1
307
308         # quitamos espacios en blanco
309         ins = ins.replace(" ", "")
310
311         # declaramos un string auxiliar para
312         # rellenarlo con 0
313         padding = ""
314
315         # rellenamos con n 0s si la instruccion
316         # aún no es de 24 bits
317         if len(ins) < 24 and not entity[-1] == ":":
318             for _ in range(0, 24 - len(ins) ):
319                 padding += "0"
320
321         # concatenamos a los 0s nuestra instruccion
322         padding += ins
323
324         # guardamos la instruccion completa
325         # en la lista a retornar
326         ans.append(padding)
327
328     return ans
```

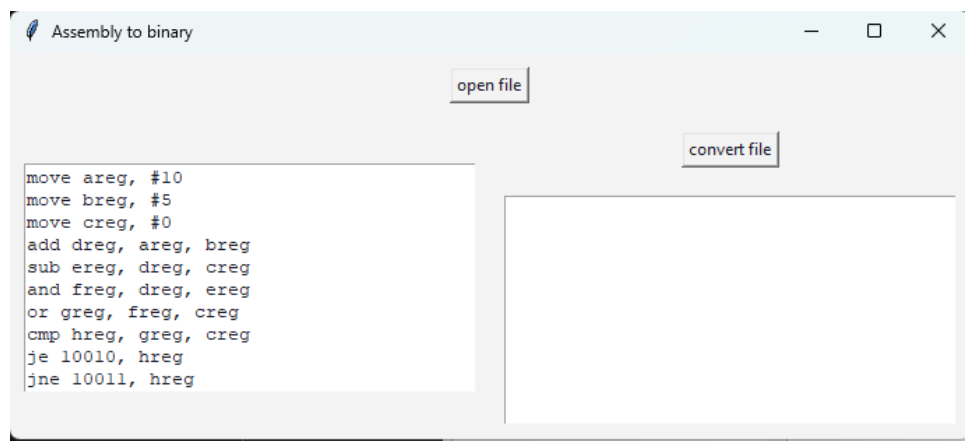
Funciones extra para la interfaz

```
331 win = tk.Tk()
332 txt_orig = tk.Text(win, wrap=tk.WORD, width=40, height=10)
333 txt_out = tk.Text(win, wrap=tk.WORD, width=40, height=10)
334 def open_file():
335     INPUT_FILE = filedialog.askopenfilename()
336     if INPUT_FILE:
337         cont = read_file(INPUT_FILE)
338         for line in cont:
339             txt_orig.insert(tk.END, line)
340
341
342 def conv_file():
343     out = conversor(INPUT_FILE)
344     for line in out:
345         txt_out.insert(tk.END, "\n")
346         txt_out.insert(tk.END, line)
347
348
349 def window() -> None:
350
351     win.title("Assembly to binary")
352
353     btn_open = tk.Button(win, text="open file", command=open_file)
354     btn_open.pack(pady=10)
355
356     txt_orig.pack(side=tk.LEFT, padx=10, pady=10)
357
358     btn_conv = tk.Button(win, text="convert file", command=conv_file)
359     btn_conv.pack(pady=10)
360
361     txt_out.pack(side=tk.RIGHT, padx=10, pady=10)
362
363     win.mainloop()
364
365
366 def main() -> None:
367
368     window()
369
370
371
372 if __name__ == "__main__":
373     main()
```

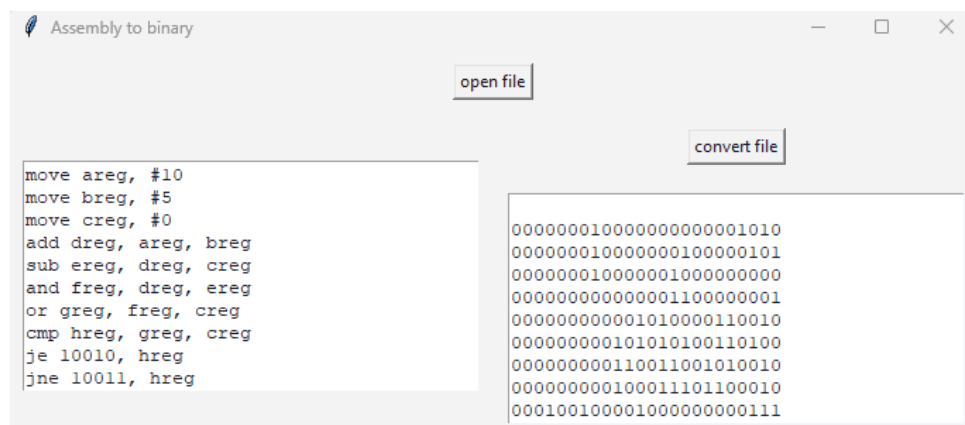
Ejecución:



Ejecución del programa

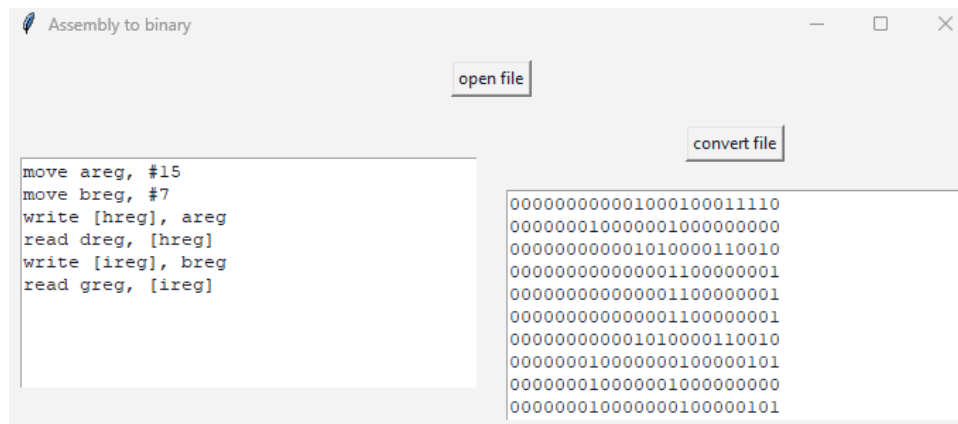


Selección del código .asm a convertir con el botón "open file"

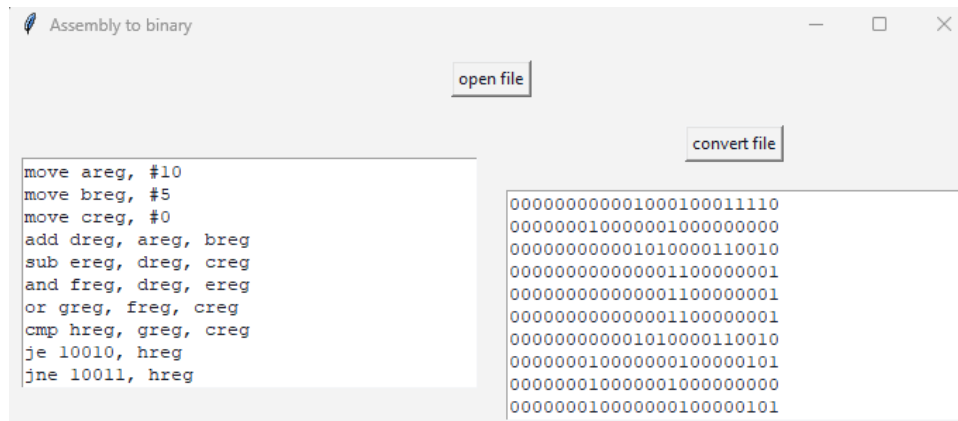


Conversión de nuestro código .asm al formato de instrucción de 24 bits

Ejemplo 1: prog1.asm



Ejemplo 2: prog2.asm



Ejemplo 3: progr3.asm

